LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-225783

Relationship between Co-op and MPI-2

David Jefferson

November 2, 2006

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Relationship between Co-op and MPI-2

David Jefferson
Oct. 30, 2006
drjefferson@llnl.gov

Summary

Several questions have naturally arisen about the relationship between Co-op (formerly known as SARS), the runtime system for the Petascale Simulation Initiative project, and MPI-2, the standard high performance message passing library. Should they be viewed as competitors or are they compatible? Are the features of Co-op essentially already present in MPI-2 or, if not, would MPI-2 be an appropriate substrate upon which to build Co-op? This short paper provides an explanation of the relationship between Co-op and MPI-2. The basic conclusions are these:

1) *MPI-2 does not directly contain the capabilities of Co-op.* Co-op is defined with higher level constructs than MPI-2 (i.e. *component* instead of *process group*, and *remote method invocation (RMI)* instead of *message passing* or *remote memory access*). Using MPI-2 directly instead of Co-op in the kinds of applications for which Co-op was designed would require, in effect, rebuilding all of Co-op's capabilities, but using MPI-2 instead of using BABEL, SLURM, TCP, and PTHREADS upon which it is built now.

2) *While it is possible, it does not seem especially desirable to build Co-op on top of MPI-2.* There are a number of software engineering considerations that make building Co-op over MPI-2 problematic.

   - Co-op would need an MPI-2 implementation with the highest level of threading support in order to build RMI from the two-sided, non-interrupting primitives provided by MPI-2. Co-op could thus only work on *high quality* MPI-2 implementations, rather than all conforming implementations.

   - RMI as defined in Co-op is fundamentally *connectionless*, whereas the MPI-2 communication primitives are mostly *connection-oriented*. In the kinds of applications for which Co-op is intended the overhead, either to have a pre-existing MPI-2 intercommunicator in place between every caller-callee pair of components, or to create and destroy an intercommunicator for every RMI call, would be large.

   - Co-op is designed so that *components* can be the units of failure and recovery. But if they use MPI-2 to communicate with one another, then a failure in one component might cause a failure in others. The MPI-2 standard does not require communicating processes to be failure-isolated from one another, and does not provide any means to kill a process group that is deadlocked, or looping.

   - Co-op is intended to support federation of *legacy* codes, with no changes to the trusted, validated algorithms in those codes. For that reason Co-op uses RMI as the basic communication primitive, in order to allow client-server relationship between

components in which neither side has to know anything about the implementation of the other. In contrast, when using MPI-2 directly, two codes generally need to be designed together and agree on a common data exchange protocol, which may entail modification to the parts of the code participating in the communication.

- A different interoperation issue arises with legacy codes if two components use different implementations of MPI-2. Since MPI-2 has no interoperation standard, it is unlikely that two such components can communicate directly without a layer of "gateway" software to translate between them. But Co-op RMI is built over Babel and TCP, both of which are designed to interoperate with *different implementations of themselves*, so interoperability between components is supported.

In general, it would be better if Co-op could be built satisfactorily over MPI-2. That would provide a shorter path to implementation of Co-op, and a more easily portable implementation as well since MPI-2 is a more uniform platform and is expected to become universally available in the scientific computing world. It would also remove Co-op's dependency on SLURM, which is primarily an LLNL-local system. But until MPI-2 is extended in some of the directions Co-op needs, it seems to be a better strategy on balance to view MPI-2 and Co-op as separate, compatible and complementary, technologies.

Detailed comparison between MPI-2 and Co-op

Introduction

MPI-2 [MPI2] is a powerful, sophisticated, and elegant collection of communication primitives designed for massively parallel and collective communication between fairly tightly-coupled processes and process groups. In that realm it has no peer, and is probably the most important software development in the last decade of scientific computing. It is based on the following ideas:

- applications composed of process groups
- either SPMD or MPMD parallelism (but easier to use with SPMD)
- two-sided, connection-oriented, blocking and nonblocking, point-to-point message communication
- two-sided, connection-oriented, massively parallel, collective communication and synchronization
- one-sided remote memory access communication (and associated synchronization)
- dynamic launch of process groups, and dynamic establishment of communication between process groups
- unification of parallel I/O with parallel communication

The Petascale Simulation Initiative project and its runtime system, Co-op, have as their goal the creation of a modest new paradigm for scalable parallel computation, especially simulation. Co-op does not attempt to create a new semantic universe for parallel computation, but rather to recombine and re-package in a novel way a number of ideas and technologies that have been

around for a long time, but have not yet been integrated into a coherent methodology. It is a high-level formalism designed around the following ideas:

- parallel, componentized applications
- SPMD parallelism within components, MPMD parallelism between them
- MPI communication within components, remote method invocation (RMI) between them
- class hierarchy and object-oriented APIs for components
- client-server relationships between components
- simple development path to componentization of legacy codes
- dynamic allocation of nodes and dynamic launch of components
- new dynamic algorithms for multiscale simulation (e.g. adaptive sampling) using dynamic launch and RMI
- component-level failure recovery

Co-op and MPI-2 represent different world views

The first things to note in a comparison between Co-op and MPI-2 is that they represent complementary views in several ways. Both are appropriate for their respective application regimes, and both should be available to application builders. But because they are very different, it should not be surprising that neither subsumes the other.

Co-op promotes a *componentized, MPMD* view of scalable parallel computation. Components run different executables, and can be dynamically launched and terminated freely. They can fail independently, without the failure spreading to other components. They are loosely-coupled, interacting in a one-sided, interrupting, connectionless, client-server style that is based on *remote method invocation (RMI)*. Components can be separately developed and still be coupled easily because they treat one another as individual *objects* (in the "object-oriented" sense) and have a uniform interaction mechanism (RMI). They present static interfaces to one another, written in Babel's SIDL language, which facilitates the incorporation of *legacy codes* into larger federations of other components that they were not originally designed to interoperate with.

The view embodied in MPI-2 is quite different. MPI-2 supports the notion of *process groups* rather than *components*. MPI-2 process groups can be dynamically created, but they cannot freely fail or be arbitrarily killed, because they are not generally isolated from each other's failures. The style of message passing communication that MPI-2 encourages is massively parallel, two-sided, connection-oriented, tightly-coupled, peer-to-peer, and usually non-interrupting. Connection-orientation generally presumes that if two process groups communicate at all, then they will communicate a lot. Two-sided communication presumes that the sequence of primitives used on one side of the communication is matched by a corresponding sequence in the other. Communicating process groups in MPI-2 do not have API's, and do not treat each other as *objects*. MPI-2 also provides one-sided remote memory access (RMA).

Co-op is defined at a higher level of abstraction than MPI-2

MPI-2 is designed to be both *low level* and *complete*. Low level means that it provides tools primitive enough that applications do not need access to still lower-level communication APIs

(e.g. UDP, platform-native packet transport libraries, etc.). *Complete* means that applications do not need any more routines at the *same* level as MPI-2 either, because MPI-2 has a sufficiently comprehensive and symmetric set of primitives that together they span the entire space of application-level high performance parallel communication needs. MPI-2 was not actually designed to be used directly by applications, but was instead intended as a base layer upon which to build higher level tools and libraries.

Co-op, on the other hand, is defined at a higher-level of abstraction. Its fundamental communication primitive, non-blocking RMI, is not very primitive at all. It has semantics similar to those of a threaded, higher-level language method call. It is non-blocking and bi-directional (call and return). Its integration with Babel provides type checking, overriding, and full exception generation. It offers the benefits of threading on both the caller and callee side; and if the body of the RMI is parallel, it can be viewed as an atomic collective. The signatures of the methods offered by a particular component can be statically published as an API, and the APIs form an inheritance hierarchy. Systems designed at a high semantic level like Co-op usually cannot aim to be "complete", at least not over the space of *all* high performance applications. Co-op is designed specifically for the class of irregular, asymmetric, asynchronous, relatively loosely-coupled, componentized applications. Many authors have noted that MPI-2 is often described as the *assembly language* of parallel communication. We can stretch that exaggerated analogy further to say that Co-op aims to be the *C++*.

Co-op is designed to support federation of legacy codes as components

One of the key goals of Co-op is to permit the easy conversion of *legacy* codes into components. We need a way of repackaging as a component a code that was designed as a standalone application, *without requiring any changes to the trusted central logic of the application.* This is the primary reason that Co-op leans so heavily on RMI as the inter-component communication mechanism, because it satisfies this requirement in two ways.

1) *For servers (RMI callees):* It allows them to make their functionality available to other components by simply adding method definitions callable from the outside. The *addition* of methods does not involve any changes to the core logic of the legacy code. And the callee of an RMI obviously does not need to know anything about the implementation of the caller.

2) *For clients (RMI callers):* It allows a subroutine or function call inside an application to be changed in place into a remote method invocation. This is a *point modification* to the code, which does not change the larger algorithm it is part of. The caller of an RMI does not need to know anything about the implementation of the callee's method; it only needs the type signature of the method.

MPI-2 is not particularly designed with federation of legacy codes in mind. If a legacy code is to communicate with another separately-designed code via MPI-2, the two codes must be modified to agree in detail on the communication protocol they will use. They have to agree not just on the types and order of data in each message (comparable to the type signature of an RMI), but also on the precise number and order of messages to be exchanged, and on the particular MPI-2

primitives they will use. Each side has to use a code pattern paralleling that used in the other. This generally forces at least some local changes to the algorithms in a legacy code, beyond a point modification. This can, of course, be ameliorated somewhat by limiting oneself to disciplined, patterned conventions for use of MPI-2 when designing communication with legacy codes.

Should Co-op be built on top of MPI?

It should be clear from the comparison so far that Co-op and MPI-2 are not positioned as competitors. Because of the difference in semantic level, it is not really appropriate to use MPI-2 directly *instead of* Co-op for the kinds of applications Co-op is intended for. Any attempt to use MPI-2 for a Co-op-like application would still require building all of the facilities of Co-op, but using MPI-2 primitives to replace the facilities used by Co-op now. But although MPI-2 is not a *substitute* for Co-op one can still ask whether it can be a *substrate* for Co-op, i.e. whether Co-op can or should be built on top of MPI-2 as a platform, instead of the combination of Babel, TCP/IP, SLURM, PTHREADS, and MPI-1 that it is built upon now.

Building on MPI-2 would certainly be desirable, because MPI-2 is a more unified platform than the collection of packages it now depends on. But in spite of its goal of completeness, MPI-2 might be missing some key feature needed by Co-op, or it might be optimized for different performance or reliability regimes than required by Co-op. After a thorough reconsideration of all aspects of the question, we am left with the following qualified conclusion:

It is technically possible, but not especially desirable, to build Co-op on top of MPI-2.

In considering how one might implement Co-op over MPI-2 there are several key issues that have to be addressed. They don't make the implementation impossible, but they make it much more complex and problematic than it might appear at first glance.

But first we should note that MPI and Co-op are *compatible* with each other, and already do coexist in federated applications. Components under Co-op are presumed to use MPI internally (either MPI-1 or MPI-2). And Co-op itself uses MPI-1 to communicate among its own daemon processes and in the mechanism for parallel RMI. Co-op does not provide any way for code in *different* components to communicate with each other via MPI; however there is no particular barrier to that either, as long as both components use the same implementation of MPI-2 and the failure tolerance consequences are acceptable.

Co-op needs a high level of threading support, which is not required by MPI-2

Co-op is based fundamentally on asymmetric (one-sided), interrupting communication using RMI. The effect of one-sided interrupting communication (without polling or busy-waiting) can only be accomplished in MPI-2 by using threading, so the relationship of MPI-2 to threading is extremely important. The MPI-2 standard only partially formalizes the relationship. (See [MPI2], Sec. 8.7.1) It requires thread safety for all MPI calls, and requires that blocking primitives block only the calling thread rather than an entire process. Beyond that the standard does not require any specific support of threading, but instead defines four *optional* levels,

MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED, and MPI_THREAD_MULTIPLE, which form a monotonic sequence of increasing support.

If threading is supported at the MPI_THREAD_FUNNELED level or higher, then any ordinary point-to-point message can effectively be made into an interrupting message by having the receiving process do the receiving call in a separate thread. The continuation of the thread after receiving a message acts as the message "handler" code. However, an implementation of MPI-2 that is to be used as a platform for Co-op must support threading at the highest level, the MPI_THREAD_MULTIPLE level, because Co-op' implementation of RMI is already heavily threaded, and requires multiple threads to be blocked concurrently. Thus, Co-op could only be built over *high-quality* implementations of MPI-2.

Co-op implements connectionless communication, but MPI-2 provides connection-oriented primitives

MPI-2 is a *connection-oriented* communication mechanism, meaning that in order to transmit data from one process to another it is required that there be a pre-existing bi-directional *connection* between them, i.e. either an *intra-communicator* or *inter-communicator*. Connection-oriented communication is appropriate when the parties are intending to exchange a large amount of data, because then the time and resource investment in setting up both sides of the connection (initializing machinery for reliability, order-preservation, flow control, synchronization, buffer management, context management, dispatching, queuing, and exception handling) can be amortized over the many transmissions that use the connection.

In contrast, RMI as defined in Co-op is *connectionless*. It does not require any connection or any pre-established relationship between sender and receiver. A thread can call a remote method out of the blue, with no warning, and the receiver is expected to be able to handle it on an *interrupting* basis. This style of communication is appropriate when the dominant character of communication between any two parties is short, infrequent, or unpredictable. It is common when there is a large fan-out or fan-in in the communication graph, but no single link is used very heavily and the pattern is statically unknown. Such patterns do not merit the time overhead and resource commitment of setting up a connection. In Co-op the design based on RMI is appropriate because the nature of the problems we had in mind (e.g. adaptive sampling) calls for scaling in the direction of many short communications with a large number of partners (high fan-in and fan-out), rather than a large volume of communication with a few partners.

We should clarify why we say that RMI is properly viewed as connectionless in Co-op, while MPI-2 is connection-oriented. The current implementation RMI uses TCP, a lower-level connection-oriented transport protocol. The connection is opened when a remote method is called, remains open during execution of the body, and is closed when the method return is complete. However, no connection between caller and callee is maintained *between* RMI calls. No separate connection setup step (open/close, or connect/accept) is required before doing an RMI call between arbitrary components. There is no contradiction in building a high-level connectionless protocol using a low-level connection-oriented protocol. For example, HTTP can be viewed as connectionless at a high level, even though it is implemented over connection-oriented TCP.

Since RMI is currently implemented over one connection-oriented protocol (TCP), one might ask if it might not just as well be implemented over a different one (MPI-2). As we view it, there are two sides to the answer.

(a) Of course RMI could, logically, be implemented over MPI-2. But we could not expect to achieve the same performance that we can with TCP. The connections used in MPI-2 (inter-communicators) are intended for communication among *process groups*, and must support all the kinds of multi-context, parallel and collective communication and synchronization primitives defined in MPI-2. In contrast, a connection created by TCP is serial, connecting just two processes, and supports only byte stream communication. Hence a TCP connection is considerably lighter-weight to establish and tear down than an intercommunicator.

(b) Furthermore, even TCP is more than Co-op needs; RMI does not fully utilize or need the TCP connection it is currently implemented with. We anticipate changing to an RMI implementation that uses a transient TCP connection for the transmission of arguments from caller to callee, and another transient connection for subsequent transmission of the returned values, with no connection at all in place during the execution of the RMI method body itself. Although this requires more setup and teardown of TCP connections, it allows a potentially much larger fan-out and fan-in of RMIs than is possible if a connection must be maintained all during the RMI execution because of the kernel-imposed limit on the number of simultaneously open connections.

Beyond the change in the implementation of RMI from one long TCP connection to two short ones, we envision eventually eliminating TCP altogether by building RMI over a truly connectionless protocol such as UDP (with reliability enhancement), or over platform-native packet transport libraries.

RMI is thus a high-level connectionless primitive. Even though our current implementation happens to use a low-level connection-oriented transport mechanism, future implementations will have a less connection-oriented substrate, or a completely connectionless one. In this respect at least, Co-op will evolve in a direction that gets farther away from implementation over MPI-2, rather than closer to it.

Co-op needs stronger failure isolation than required by MPI-2

Co-op is designed with the goal that components can fail independently, or be killed, and then be restarted, possibly from a component-level checkpoint if there is one, without bringing down the entire federated computation. Our intent is that the *component* should be the unit of failure recovery.

The MPI-2 standard, although it does provide facilities for launching new process groups, does not provide facilities for safe abnormal termination of, or killing of, process groups. Processes or process groups generally cannot terminate abnormally without the risk of bringing down the rest of the MPI-2 computation. Processes or process groups that are deadlocked or looping likewise

cannot be killed within MPI-2, and cannot be safely killed even by external mechanisms without risking the rest of the computation. The MPI-2 standard is clear that failure isolation across an inter-communicator is not required. ([MPI2], p. 95, ll 36-41). To be used for an implementation of Co-op, either (a) the MPI-2 implementation would have to go well beyond the standard and provide some failure isolation facilities, or (b) we would need to add another layer of process control and insulating software over MPI-2 to provide for fault isolation and recovery, or (c) we would have to give up the goal of Co-op being able to handle failure and recovery at the component level.

Some implementations of MPI-2, e.g. those based on the experience of FT-MPI, may go beyond the standard and provide process group abortion and failure-handling facilities. But if Co-op were to make essential use of extensions of MPI-2 that go beyond the standard then, as is the case with threading support, it might not be portable to other MPI-2 implementations.

Can Co-op use MPI-2 for part of its functionality?

Co-op offers two fundamental services that MPI-2 overlaps with: component management (launch and shutdown), and RMI. For reasons described above, it is not attractive to build RMI from MPI-2. But if we could build the component launch capability from the MPI-2 *spawn()* primitive that launches new process groups, then we could at least eliminate our dependence on SLURM and be more portable to other environments.

Unfortunately, the MPI-2 *spawn()* primitive is not close enough to what Co-op needs. When a parent process group spawns a child process group, an intercommunicator between parent and child is always created. But in Co-op we do not really want that intercommunicator because (i) it would not be used for RMI, (ii) its creation adds unwanted overhead (though presumably a small amount compared to the rest of the time for *spawn()*), and (iii) it may entangle the fate of the child with that of the parent in the sense that if one fails, the other may also. The MPI-2 committee considered defining a version of the *spawn()* primitive that does not create an intercommunicator, for use in those occasions when the parent does not intend to communicate with the child; but that proposal did not become part of the standard. (See [MPI2JOD], Ch. 2.)

SLURM also provides some component and process control services that MPI-2 does not. It monitors the execution of components after they are launched. If a component terminates, or one of its processes fails, SLURM cleans up all processes in the component and notifies Co-op and the parent component so that they can take appropriate action. If we abandoned SLURM to use MPI-2, we would have to rebuild these facilities some other way.

MPI-2 implementations do not necessarily interoperate with each other

The MPI-2 standard is not (yet) accompanied by an *interoperation* standard. There is no guarantee that two process groups using different implementations of MPI-2 can directly create an intercommunicator between them and exchange MPI-2 messages. If a process group attempts a *spawn()* operation to launch a new process group and create an intercommunicator between the parent and child, we cannot expect that to work unless the parent and child codes are built with the same implementation of MPI-2. Likewise, if two codes are already executing, and attempt to

establish an intercommunicator using MPI-2 ports and the MPI-2 *accept()* and *connect()* calls, that also requires that both sides of the connection be using the same (or compatible) implementations of MPI-2. This limitation could become increasingly important if multiple MPI-2 implementations coexist in the "market" and the differences between them are important enough that neither dominates. Then we can expect different codes to be built around different MPI-2 implementations, and coupling them with MPI-2, or a mechanism built using MPI-2, may become problematic. We note that there is an interoperability standard for MPI-1 (see [IMPI]), but so far none for MPI-2.

Co-op is specifically intended to allow *legacy* codes, and codes separately designed from one another, to be easily federated into a single application. They should be able to interoperate even if they use different implementations of the inter-component communication software. For this reason, the Co-op RMI mechanism is built using Babel and TCP/IP, both of which are designed specifically for interoperability *with different implementations of themselves*. But a Co-op implementation built directly on top of MPI-2 could not implement RMI in a way that would work between *arbitrary* legacy codes without a layer of "gateway" code to translate the representations and conventions of each MPI-2 implementation to those of the others.

Conclusion

Co-op and MPI-2 are defined at different levels of abstraction, so neither is a substitute for the other. If we instead consider building Co-op over MPI-2, we then encounter a number of software engineering issues that are sufficiently difficult to overcome that it seems best to use an alternate implementation strategy. Perhaps MPI-2 can be extended some day in the directions needed by Co-op. It would then provide a shorter path to implementation of Co-op, and a more easily portable implementation as well. And I believe it would also improve MPI. But until then, Co-op and MPI-2 are perhaps better thought of as *complementary* technologies.

Acknowledgments

This document has benefited greatly from the comments of John Tannahill, Gary Kumfert, and especially Bronis de Supinski.

References

[IMPI] IMPI Steering Committee, “IMPI: Interoperable Message Passing Interface”, January 2000 (Available at ftp://ftp.nist.gov/pub/hpss/interop/impi-report.current.ps)

[MPI2] Message Passing Interface Forum, “Extensions to the Message Passing Interface”, July 18, 1997. Available at http://www.mpi-forum.org/docs/docs.html

[MPI2JOD] Message Passing Interface Forum, “MPI-2 Journal of Development”, July 18, 1997. Available at http://www.mpi-forum.org/docs/docs.html